Comencemos a programar con

VBA - Access

Entrega 15

Operadores

Eduardo Olaz

Operadores

A la hora de construir instrucciones en VBA, que contengan operaciones, no sólo manejamos constantes, variable y expresiones, sino que utilizamos unos elementos llamados Operadores, que aplicados a uno ó varios operandos, generan el resultado de la operación.

Tipos de Operadores

Aritméticos	Se usan para efectuar cálculos matemáticos
De Comparación	Permiten efectuar comparaciones
De Concatenación	Combinan cadenas de caracteres
Lógicos	Realizan operaciones lógicas

Operadores aritméticos

VBA maneja la mayor parte de los operadores aritméticos habituales en los lenguajes de programación.

Estos operadores son

- **+** Suma
- **−** Resta
- ***** Producto
- **/** División
- **^** Elevar a potencia
- **** División entera
- **Mod** Módulo ó Resto

En general, el tipo devuelto por el resultado de una operación, es el del tipo del más preciso de los operadores, salvo que el resultado supere su rango; en ese caso devolverá el siguiente tipo de mayor precisión. Esta regla tiene abundantes excepciones.

Para más información consulte la ayuda de Access en la que se relata toda la casuística pormenorizada para cada uno de los operadores.

Si el resultado de una operación fuese un dato de coma flotante y se asignara a un tipo entero, se efectuaría un redondeo.

Si se trata de asignar un resultado fuera del rango de valores de la variable que va a recibir el resultado de la operación, se generará un error de "Desbordamiento" y se interrumpirá la ejecución del código, salvo que el error fuera capturado y tratado.

Operador Suma

El operador **suma** (**+**), sirve para asignar el resultado de la suma de dos números, ó en el caso de cadenas, dar como resultado una cadena compuesta por las dos anteriores.

La forma de usarlo es

```
resultado = expresión1+expresión2
```

Si *expresión1* ó *expresión2* tuvieran el valor **Null**, el resultado sería también el valor **Null**.

expresión1 y ***expresión2*** son los operandos, pudiendo ser cualquier valor numérico ó expresiones que los generen.

Al contrario de otros lenguajes, VBA permite utilizar como operadores, tipos numéricos distintos. Por ejemplo podemos sumar un tipo **Byte** con un tipo **Long**, ó con un tipo **Date**.

Igualmente la variable ***resultado*** no tiene por qué ser del mismo tipo que los operandos.

Una de las limitaciones es que el resultado de la operación no debe sobrepasar la capacidad del tipo correspondiente a la variable que va a recibir el resultado de la misma.

Esto es aplicable al resto de los operadores.

Por ejemplo

```
Public Sub SumaConError()

    Dim bytResultado As Byte

    bytResultado = 10 + 23
    Debug.Print bytResultado
    bytResultado = 150 + 150
    Debug.Print bytResultado
End Sub
```

Nos imprimirá el resultado 33 y a continuación nos generará el error nº 6 "Desbordamiento", ya que un tipo Byte sólo admite valores que van de 0 a 255.

Nos daría ese mismo tipo de error si tratáramos de hacer

```
bytResultado = 15 + (-16)
```

Ya que a un **Byte** no se le pueden asignar valores negativos

Lo mismo ocurriría en el siguiente código

```
Dim intResultado As Integer

    intResultado = 30000 + 30000
```

ya que un tipo **Integer** maneja valores entre **-32.768** y **32.767**.

Como hemos indicado en un punto anterior, si a una variable que maneja números enteros le asignamos el resultado de una suma de números de coma flotante, efectuará un redondeo del resultado al número entero más próximo.

```
Public Sub PruebaSumaComaFlotante()
    Dim intResultado As Integer
    intResultado = 3.1416 + 2.5468
    Debug.Print intResultado
    intResultado = -3.1416 + (-2.5468)
    Debug.Print intResultado
End Sub
```

Nos mostrará como resultado los valores **6** y -**6**.

Cuando se utilizan como operandos dos valores de tipos distintos, VBA cambia el menos preciso al tipo del más preciso. Por ejemplo si vamos a sumar un **Integer** con un **Long**, VBA realiza un "moldeado de tipo" con el **Integer**, convirtiéndolo a **Long** antes de realizar la operación.

Si uno de los operadores fuera del tipo **Date**, el resultado también lo será.

Si desde la ventana Inmediato hacemos

```
? #3/14/5# + 1
```

nos mostrará

```
15/03/2005
```

(El comando **?** es equivalente a **Print**)

En este caso vemos que al sumar **1** a la fecha devuelve la fecha del día siguiente.

Como ya hemos comentado el operador Suma permite sumar o concatenar cadenas.

Esta facilidad puede en algún momento crearnos más problemas que ventajas sobre todo cuando manejamos cadenas que contienen posibles valores numéricos.

```
Public Sub PruebaSumaCadenas()
    Dim strResultado As String

    strResultado = 3 + 4
    Debug.Print strResultado

    strResultado = 3 + "4"
    Debug.Print strResultado

    strResultado = "3" + "4"
    Debug.Print strResultado
End Sub
```

Este código nos devolverá

```
7
7
34
```

Curiosamente `3 + "4"` devuelve 7.

Además si tratamos de hacer

```
101 + " dálmatas"
```

Nos dará el "bonito error" nº 13; "No coinciden los tipos".

Por ello recomendamos usar el operador **&** como operador para sumar, ó unir cadenas, en vez del operador Suma **+**.

```
101 & " dálmatas" → "101 dálmatas"
3 & "4" → "34"
```

Hay otro sumando permitido por VBA que no deja de ser sorprendente. Es el valor **Empty** (vacío) que si lo usamos con un número, ó consigo mismo, se asimila al **Cero**. Y con una cadena a la cadena vacía "".

```
Empty + "A"   →   "A"
Empty + 3.1416   →   3.1416
Empty + Empty  →   0
```

Esta "promiscuidad" en el manejo y la asignación entre tipos diferentes de datos que permite Visual Basic para Aplicaciones, es algo que personalmente no me termina de convencer, pero como decía el castizo, - "es lo que hay…".

VBA no permite los operadores del tipo Pre ó Post Incremental como serían:

```
Y = ++X   ó   Y = X++
```

Operador Resta

El operador **resta** (**-**), sirve para asignar el resultado de la sustracción entre dos números.

Tiene dos formas sintácticas

```
resultado = expresión1-expresión2
- expresión
```

En la primera, la variable ***resultado*** recibe el valor resultante de restar ***expresión2*** a ***expresión1***.

En la segunda se cambia el signo al valor numérico de ***expresión***.

Si ***expresión1*** ó ***expresión2*** tuvieran el valor **Null**, ***resultado*** recibirá también el valor **Null**.

Empty lo considera como valor **Cero**.

Como en el caso de la suma, si uno de los operadores fuera del tipo **Date**, el resultado también lo será. Si desde la ventana Inmediato hacemos:

```
Print #3/14/5# - 100
```

nos mostrará

```
04/12/2004
```

Para obtener información sobre los distintos tipos devueltos, en función del de los tipos de los operadores, consulte la ayuda de VBA.

Operador Producto

El operador **producto** (*), sirve para asignar el resultado del producto de dos números.

La forma de usarlo es

```
resultado = expresión1*expresión2
```

resultado es una variable de tipo numérico y tanto ***expresión1*** como ***expresión2*** pueden ser cualquier expresión que de como resultado un valor numérico.

El tipo numérico que se suministra a *resultado* dependerá de los tipos de *expresión1* y *expresión2*.

Si uno de los operandos es del tipo **Single** y el otro del tipo **Long**, ***resultado*** recibirá un tipo **Double**. Para más información consultar la ayuda de VBA.

Si alguno de los operadores es **Null**, ***resultado*** tomará también el valor **Null**.

Si alguno de los operandos es **Empty**, ***resultado*** será **Cero**. Ya que considerará que ese operando contiene el valor **Cero**.

Operador División

El operador **división** (**/**),asigna el resultado de la división de dos números.

La forma de usarlo es

resultado **=** *expresión1***/***expresión2*

resultado es una variable de tipo numérico y tanto ***expresión1*** como ***expresión2*** pueden ser cualquier expresión que de como resultado un valor numérico.

Si ***expresión2*** fuera el valor Cero, daría el error **11** de "División por cero".

El tipo numérico que recibe *resultado* normalmente será del tipo **Double**.

Esta regla tiene varias excepciones

Si los operandos son del tipo **Byte**, **Integer** ó **Single**, ***resultado*** recibirá un **Single**, a menos que supere el rango de **Single** en cuyo caso será del tipo **Double**.

Si uno de los operandos fuera del tipo **Decimal**, resultado también lo será.

Es aplicable lo dicho en los anteriores operadores matemáticos respecto a **Null** y **Empty**.

Para más información consultar la ayuda de VBA.

Operador Elevar a potencia

El operador para **elevar a potencia** (**^**),asigna el resultado de elevar la base a la potencia del exponente.

La forma de usarlo es

resultado **=** *expresión1***^***exponente*

resultado es una variable de tipo numérico y tanto ***expresión1*** como ***exponente*** pueden ser cualquier expresión que de como resultado un valor numérico.

Si ***exponente*** fuera el valor **Cero**, daría como resultado **1** , incluso en el caso de que la base tuviera el valor **Cero**. **¿¿¿???**.

```
45^0 → 1
0^0  → 1      (algo no muy correcto matemáticamente hablando)
3.1416 ^ Empty → 1
Null ^ 0 → Null
3.1416 ^ Null → Null
```

Exponente puede ser un valor fraccionario.

Así, si quisiéramos obtener la raíz cúbica de 343 podríamos hacer

```
343 ^ (1/3)  → 7
```

Actuaríamos igual para la raíz cuadrada

```
16 ^ 0.5   →    4
```

Adicionalmente, para obtener la raíz cuadrada en VBA existe la función **Sqr**.

```
Sqr(16)   →    4
```

Cuando estudiábamos matemáticas nos contaron que en el campo de los números reales, la raíz cuadrada de un número negativo no tiene solución. Por ello se crearon los números imaginarios.

Si tratamos de hacer **Sqr**(-16)nos generará un error 5 en tiempo de ejecución, indicando que el valor -16 no es correcto para la función **Sqr**.

Puede sorprender que esto no ocurra si calculamos la raíz cuadrada elevando a 0.5

```
-16 ^ 0.5   →   -4
```

En cambio si hacemos (-16)^0.5, sí dará el error.

La razón la veremos más adelante en la prioridad de operadores.

El cálculo del operador potencia se realiza antes que el del operador de cambio de signo.

Tanto en la base como en el exponente admite números de coma flotante.

Por ejemplo podríamos obtener algo tan exótico como el resultado de elevar el número **e**, base de los logaritmos Neperianos, al valor de **Pi**

```
2.7182818 ^ 3.1415926   →   23,1406906315563
```

Operador División entera

El operador **división entera** (****), realiza dos procesos.

Si no tuvieran valores enteros, efectuaría el redondeo del numerador y del denominador.

A continuación realiza la división de los valores resultantes, devolviendo la parte entera del resultado.

La forma de usarlo es

```
resultado = expresión1\expresión2
```

resultado recibirá un valor entero Byte, Integer ó Long.

```
8 \ 3   →   2
-8 \ 3   →   -2
12 \ 3   →   4
```

Es aplicable lo comentado con anterioridad respecto a **Empty** y **Null**.

```
Null \ 3   →   Null
Empty \ 3   →   0
```

Si al redondear ***expresión2*** diera como resultado el valor Cero, se produciría el error **11** de “División por cero”.

```
4 \ 0.1   → Error 11
4 \ 0.5   → Error 11
4 \ 0.51   → 4
4 \ Empty   → Error 11
```

```
4 \ 0   → Error 11
```

Hay que tener un especial cuidado con este operador, y no olvidar que previamente realiza un redondeo a cero decimales, tanto del numerador como del denominador.

Este **redondeo** utiliza el método de redondeo vigente en el **sistema bancario americano**, que es ligeramente diferente al europeo.

De este tema hablaremos cuando veamos la función **Round**.

Operador Módulo o Resto

El operador **módulo** (**Mod**) , Asigna el resto de una división entre dos números.

Como en el caso de la división entera, previamente se realiza un redondeo a cero decimales, tanto del dividendo como del divisor, si éstos tuvieran un valor que no fuera entero.

Su sintaxis es

```
resultado = expresión1 Mod expresión2
```

Para el operador **Mod**, es aplicable lo comentado en el operador División entera sobre el redondeo del numerador y denominador.

```
8 Mod 7    →    1
8.6 Mod 7    →    2
8.9 Mod 7.51    →    1
Null Mod 7.51    →    Null
Empty Mod 7.51    →    0
27 Mod 12    →    3
```

Operadores de Comparación

En algunos procesos podemos necesitar establecer si un valor es menor, igual ó mayor que otro, por ejemplo para la toma de una decisión mediante una sentencia **If . . . Then**.

En VBA tenemos 6 tipos de operadores para esta tarea.

Estos operadores son

- **<** Menor que
- **<=** **=<** Menor ó igual que
- **>** Mayor que
- **>=** **=>** Mayor ó igual que
- **=** Igual a
- **<>** **><** Distinto de

Vemos que para los operadores **menor o igual**, **mayor o igual** y **distinto de** hay dos posibles formas de escritura. La forma más usual de uso es la escrita en primer lugar.

Su sintaxis es

```
resultado = expresión1 Operador expresión2
```

resultado es una variable de tipo **boleano** y tanto ***expresión1*** como ***expresión2*** pueden ser cualquier expresión que de como resultado un valor numérico o de cadena.

Si ***expresión1*** o ***expresión2*** contuvieran el valor **Null** devolvería **Null**.

Ejemplos de resultados

```
8 < 7      →   False
7 < 8      →   True
8 <= 8     →   True
8 >= 8     →   True
8 > 7      →   True
8 = 8      →   True
8 <> 8     →   False
"Avila" < "Barcelona"  →   True
```

A la hora de comparar cadenas, hay que tener en cuenta que el resultado puede cambiar en función de cómo hayamos establecido la forma de comparación en

```
Option Compare
```

Por ejemplo, si no hemos establecido, a nivel del módulo, valor para **Option Compare**, o hemos puesto **Option Compare Binary**, las siguiente expresiones darán **False**.

```
"A"="a"   →   False
"Ávila" < "Barcelona"   →   False
```

En cambio si tenemos **Option Compare Text**, las siguientes expresiones darán **True**.

```
"A"="a"   →   True
"Ávila" < "Barcelona"   →   True
```

Si utilizamos **Option Compare Database**, (sólo con bases de datos Access) dependerá de cómo hemos definido la comparación a nivel local en la base de datos.

Todo lo anterior es aplicable para el resto de los operadores de comparación con cadenas de texto.

Operador Like

En la entrega anterior vimos la función **InStr**, que permite buscar una cadena dentro de otra y nos devuelve la posición en la que se encuentra. También vimos la función **StrComp** que compara dos cadenas diciéndonos cuál es “menor”.

Para la comparación entre cadenas tenemos también un operador adicional

- **Like** Permite comparar una expresión patrón con una cadena, indicándonos si esa cadena cumple con lo indicado en la patrón.

Su sintaxis es

```
resultado = cadena Like patrón
```

Para el operador **Like**, también es aplicable lo comentado anteriormente sobre **Option Compare**. Para obtener más información al respecto puede consultar la información sobre el operador **Like** en la ayuda de VBA.

Si el contenido de *cadena* coincide con el contenido de *patrón*, ***resultado*** recibirá el valor **True**.

En caso contrario se le asignará False, salvo que alguno de los operandos ***cadena*** o ***patrón***, tenga como valor **Null**, en cuyo caso se le asignará **Null**.

El parámetro **patrón** se construye con una serie de caracteres, algunos de los cuales pueden funcionar como "comodines".

Supongamos que queremos ver si una cadena comienza por "MAR"

```
"MARTINEZ ELIZONDO" Like "MAR*"  →  True
```

En este caso **"MAR*"** devuelve True, porque "MARTINEZ ELIZONDO" comienza por **"MAR"** . El carácter **"*"** sustituye a cualquier conjunto de caracteres.

Incluso **"MAR" Like** "MAR*" ó **"MAR-2" Like** "MAR*" también devolverían **True**.

Hay otros dos caracteres comodines.

"?" sustituye a un carácter cualquiera

"#" sustituye a un dígito cualquiera

```
"MARTA" Like "MART?"  →  True
"MART2" Like "MART?"  →  True
"MART24" Like "MART?"  →  False
"MARTA" Like "MART#"  →  False
"MART2" Like "MART#"  →  True
"MART24" Like "MART#"  →  False
```

Podemos ver si un carácter está dentro de un intervalo de caracteres, mediante **Listas de caracteres**. Las listas de caracteres se especifican escribiendo los caracteres con los que queremos comparar el carácter, entre corchetes, empezando el patrón con comillas.

Hay varias formas de construir una lista de caracteres.

Caracteres sueltos entre comas

```
"a" Like "[a,b,c,d]"  →  True
"n" Like "[a,b,c,d]"  →  False
```

Una cadena de caracteres

```
"a" Like "[abcd]"  →  True
"n" Like "[abcd]"  →  False
```

Podemos definir rangos de caracteres indicando el carácter inferior y el superior separados por un guión.

```
"a" Like "[a-d]"  →  True
"n" Like "[a-d]"  →  False
```

Podemos combinar caracteres sueltos y rangos separándolos por comas.

Supongamos que queremos comprobar si un carácter está entre la **"a"** y la **"d"** o entre la **"m"** y la **"q"**, considerando también como carácter válido la **"ñ"**.

```
"a" Like "[a-d,m-q,ñ]"  →  True
"o" Like "[a-d,m-q,ñ]"  →  True
"ñ" Like "[a-d,m-q,ñ]"  →  True
```

```
"e" Like "[a-d,m-q,ñ]" → False
"1" Like "[a-d,m-q,ñ]" → False
```

Podemos combinar en las cadenas patrón, caracteres, comodines y listas lo que nos permite definir patrones complejos.

Supongamos que queremos que nos de **True** si la cadena chequeada cumple con estas condiciones:

1. Empieza por una letra entre la **"C"** y la **"H"**, o entre la **"Q"** y la **"S"**.
2. El segundo carácter debe ser un número
3. El tercer carácter puede ser cualquiera
4. Los cuatro siguientes caracteres deben ser **"00BB"**
5. El 8º carácter no debe ser la letra V. (Si utilizamos el carácter Exclamación **"!"** equivale a la negación de lo que le sigue).

Vayamos por partes y construyamos los componentes de la cadena patrón.

1. **"[Q-R]"**.
2. **"#"**.
3. **"?"**.
4. **"00BB"**
5. **"[!V]"**.

Juntándolo nos quedará **"[C-H,Q-R]#?00BB[!V]*"**.

```
"R5Ñ00BBS" Like "[C-H,Q-R]#?00BB[!V]*" → True
"R5Z00BBS-12345" Like "[C-H,Q-R]#?00BB[!V]*" → True
"Q8W00BBAACC" Like "[C-H,Q-R]#?00BB[!V]*" → True
```

De lo aquí expresado podemos ver que el operador **Like** permite efectuar comprobaciones ó validaciones de cadenas verdaderamente complejas y con una sola línea de código.

Nota:

A la hora de definir una lista de rangos del tipo [**A-H**], el primer carácter debe ser menor que el segundo. Por ejemplo [**H-A**] no sería un patrón válido.

En la ayuda de VBA se realiza una descripción pormenorizada de toda la casuística con la que nos podemos encontrar.

Operador Is

Para la comparación entre objetos tenemos un operador adicional, es el operador **IS**.

- **Is** devuelve **True** o **False** en función de que dos variable objeto hagan referencia al mismo objeto.

Este tema lo veremos cuando veamos más a fondo los Objetos.

Hay otras 2 formas de utilizar el operador **Is**.

La primera ya la vimos con la estructura **Select . . . Case**

Lo usamos, por ejemplo en las expresiones del tipo

```
Select Case N
Case is < 10
     - - -
```

```
Case 10
     - - -
Case is > 10
```

En ellas comprobamos si la variable pasada como testigo es menor ó mayor que 10.

Hay otro entorno en el que se usa **Is** y es en estructuras del tipo **If** . . . **Then**; en concreto acompañando a la expresión **TipeOf**.

TipeOf devuelve el tipo de un objeto.

Se usa de la siguiente forma

TypeOf *nombre_objeto* **Is** *tipo_objeto*

Esto nos puede servir, por ejemplo para cambiar las propiedades de objetos en función del tipo al que pertenecen.

Para ilustrar esto vamos a crear un formulario nuevo.

Al cargar el formulario, examinará los controles que hay en él.

Asignará a todos los controles la misma altura (propiedad **Height**) y ordenará las etiquetas, cuadros de texto y botones que contenga.

Ajustará la distancia que haya entre la parte izquierda del control y la parte izquierda del formulario (propiedad **Left**).

Ajustará también la distancia respecto a la parte superior de la sección donde están ubicados (propiedad **Top**).

Las otras dos propiedades que ajustará serán

Texto del control (propiedades **Caption** o el valor por defecto de los cuadros de texto)

Anchura del control (propiedad **Width**).

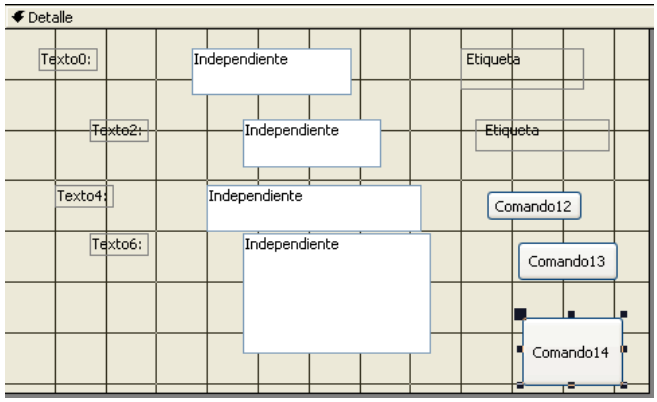
En la barra del menú del Cuadro de Herramientas, desactivamos el botón **[Asistentes para controles]** (la de la varita mágica).

Ponemos en el formulario

- 4 Cuadros de texto
- 2 Etiquetas adicionales (ponles un texto para ver el efecto)
- 3 botones

Procuraremos poner los controles de una manera desordenada, y variando sus tamaños.

Nuestro formulario podría tener un aspecto tan patético como éste:



Vemos que no es probable que obtenga el premio al diseño del formulario del año.

Para tratar de poner orden en este caos, vamos a hacer que sea Access el que lo haga utilizando código. Aprovecharemos el evento **Al cargar** del formulario

Teniendo seleccionado el formulario, en la ventana de propiedades seleccionamos el evento Al cargar y en el editor de código escribimos lo siguiente:

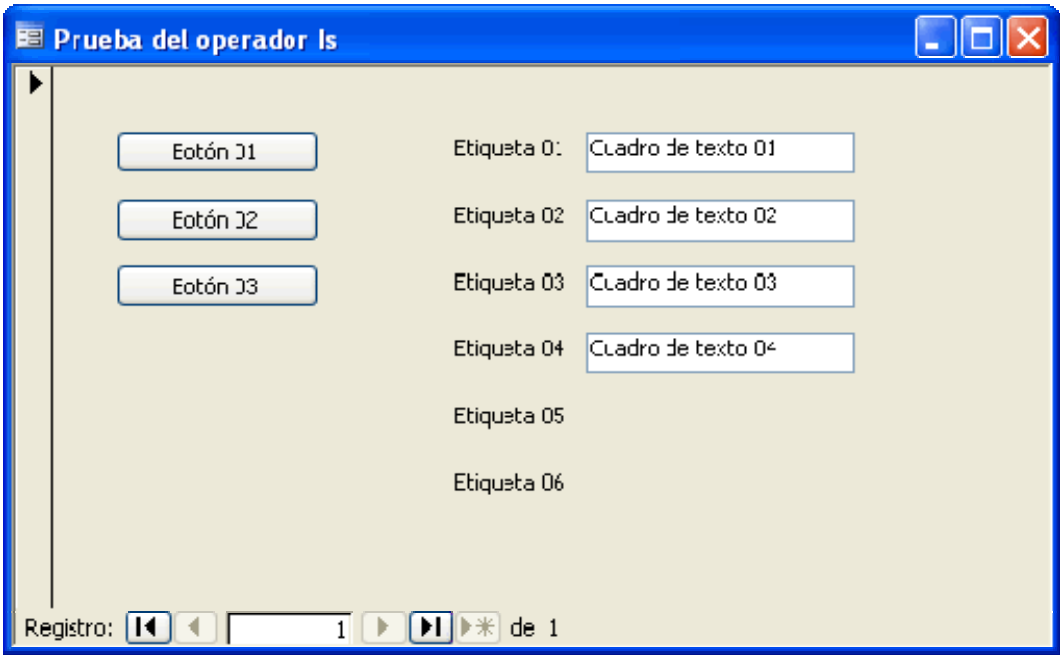
```
Private Sub Form_Load()
    Const conlngAltura As Long = 300
    Dim ctr As Control
    Dim lngEtiquetas As Long
    Dim lngCuadrosTexto As Long
    Dim lngBotones As Long

    Caption = "Prueba del operador Is"
    For Each ctr In Me.Controls
        ctr.Height = conlngAltura

        If TypeOf ctr Is CommandButton Then
            ' Si el control es un botón
            lngBotones = lngBotones + 1
            ctr.Left = 500
            ctr.Top = lngBotones * 500
            ctr.Width = 1500
            ctr.Caption = "Botón " _
                    & Format(lngBotones, "00")
        End If
        If TypeOf ctr Is Label Then
            ' Si el control es una etiqueta
            lngEtiquetas = lngEtiquetas + 1
            ctr.Left = 3000
            ctr.Top = lngEtiquetas * 500
            ctr.Width = 2500
            ctr.Caption = "Etiqueta " _
                    & Format(lngEtiquetas, "00")
        End If
        If TypeOf ctr Is TextBox Then
            ' Si el control es un cuadro de texto
            lngCuadrosTexto = lngCuadrosTexto + 1
            ctr.Left = 4000
            ctr.Top = lngCuadrosTexto * 500
            ctr.Width = 2000
            ctr = "Cuadro de texto " _
                    & Format(lngCuadrosTexto, "00")
```

```
            End If
        Next ctr
    End Sub
```

Este código hace que el formulario, tras cargarse, tome este otro aspecto



Podemos decir que esto ya es otra cosa.

En vez de utilizar una sentencia `If TypeOf ctr Is Control` podríamos haber utilizado una de las propiedades que tienen los controles que es **ControlType**.

```
Ctr.ControlType
```

nos hubiera devuelto una constante enumerada del tipo **acControlType**. En concreto, y en este caso, **acLabel**, **acTextBox** o **acCommandButton**.

Hay una función adicional que tiene relación con **TypeOf**. Es la función **TypeName**.

TypeName devuelve una cadena con información acerca de la variable que se le pase como parámetro.

Su sintaxis es **TypeName(***nombrevariable***)**

La variable ***nombrevariable*** puede ser de prácticamente cualquier tipo.

En nuestro caso, vamos a añadir la línea escrita en negrita al código anterior

```
For Each ctr In Me.Controls
        ctr.Height = conlngAltura
        Debug.Print TypeName(ctr)
```

Este código nos irá imprimiendo los tipos de controles que va examinando; en concreto las cadenas `Textbox`, `Label` y `CommandButton`.

Dejo al lector el desarrollo de un código equivalente al visto, pero utilizando la propiedad **ControlType** o la función **TypeName**.

Tanto **ControlType** como **TypeOf** ctr **Is** Control o **TypeName** tienen utilidad, por ejemplo para asignar dinámicamente propiedades a controles y evitar aquéllas propiedades que no pertenecen al control con el que se esté trabajando.

Esto sucede por ejemplo, en aplicaciones multi idioma, en las que queremos adaptar los textos de las diferentes Interfaces al idioma seleccionado por el usuario, de forma dinámica.

Otra ejemplo de aplicación sería diseñar formularios ajustables a la resolución que utilice el usuario en su pantalla.

Operadores de concatenación

Los operadores de **concatenación** sirven para unir varias cadenas en una sola.

Estos operadores son

- **+** Suma
- **&** Ampersand

Ya he comentado la posibilidad que tiene el operador suma para unir cadenas, al igual que el operador **&**.

También he comentado mi preferencia por el operador **&** para efectuar estas tareas.

La sintaxis para estos operadores es

```
resultado = expresión1 + expresión2
resultado = expresión1 & expresión2
```

Operadores lógicos

Los operadores **lógicos** sirven para realizar operaciones lógicas con los operandos.

Estos operadores son

- **And** Conjunción Lógica
- **Or** Disyunción lógica
- **Xor** Exclusión lógica
- **Not** Negación lógica
- **Eqv** Equivalencia lógica
- **Imp** Implicación lógica

Los más usados son los cuatro primeros, **And**, **Or**, **Xor** y **Not**.

Los seis operadores, además de trabajar con expresiones de tipo lógico

```
Resultado = ExpresionBooleana Operador ExpresionBooleana
```

permiten realizar comparaciones a nivel de bit con operandos numéricos.

Los operandos puede tener el valor **Null**, en cuyo caso el resultado dependerá de si es el primero, el segundo ó los dos.

Operador And

El operador **And**, realiza una conjunción lógica entre dos expresiones operandos.

Su sintaxis es

resultado **=** *expresión1* **And** *expresión2*

Resultado puede ser cualquier variable numérica ó boleana.

resultado tomará el valor **True**, sólo y sólo si las dos expresiones son ciertas.

Las combinaciones de **Null** con **True** dan **Null**.

Null con **Null** da **Null**

Las combinaciones que contengan **False** dan **False**.

Cuadro de resultados del operador **And**

Expresión 1ª	Expresión 2ª	Resultado
True	True	**True**
True	False	**False**
True	Null	**Null**
False	True	**False**
False	False	**False**
False	Null	**False**
Null	True	**Null**
Null	False	**False**
Null	Null	**Null**

Veamos las siguientes expresiones

```
8 > 4 And 4 >= 3     →     True
IsNumeric("1234") And Len("1234") = 4     →     True
IsNumeric("Pepe") And Len("1234") = 4     →     False
```

Las operaciones de **And** con números se realizan a nivel de Bits.

```
7 And 13   →   5
```

¿Cómo puede ser esto?

7, en binario es igual a 0**1**1**1**

13 es igual a 1**1**0**1**

El resultado será 0**1**0**1**

El binario **1**0**1** equivale en notación decimal al número **5**

And devuelve el bit **1** sólo si los dos bits correspondientes de los operandos, valen **1**.

Este operador es útil para poder averiguar si un determinado bit de una expresión está activado (tiene el valor 1).

Por ejemplo, si queremos averiguar directamente el valor del tercer bit de un número entero **M**, es suficiente ver el resultado de realizar **M And 4**. Si el resultado es **4** el tercer bit de **M** contiene el valor 1, si es 0 el tercer bit contendrá el valor 0.

En general si queremos averiguar el bit Nº **N** de una expresión **M** comprobaremos si el resultado de **M And 2^(N-1) es igual** a **2^(N-1)**.

Hay determinados dispositivos en los que la configuración viene determinada por los bits de una determinada propiedad.

Operador Or

El operador **Or**, realiza una Disyunción lógica entre dos operandos.

Su sintaxis es

resultado **=** *expresión1* **Or** *expresión2*

Resultado puede ser cualquier variable numérica ó boleana.

resultado tomará el valor **True**, si cualquiera de las dos expresiones es **True**.

Las combinaciones de **False** con **False** da **False**.

Null con **False** ó con **Null** da **Null**

Cuadro de resultados del operador **Or**

Expresión 1ª	Expresión 2ª	Resultado
True	True	**True**
True	False	**True**
True	Null	**True**
False	True	**True**
False	False	**False**
False	Null	**Null**
Null	True	**True**
Null	False	**Null**
Null	Null	**Null**

A nivel de bits, en operaciones con números, **Or** dará **1** salvo en el caso de que ambos bits valgan **0**.

```
5 Or 13  →  13
         0101
         1101
         1101
```

El binario **1101** es en notación decimal el número **13**

Con el operador **And**, hemos visto que podemos averiguar qué bit está activado en una expresión.

El operador **Or** nos permite definir el valor de un determinado bit de un valor.

Supongamos que tenemos un dispositivo que contiene la propiedad **Config** de tipo **Long**, que controla el funcionamiento del mismo.

Por circunstancias de la programación queremos hacer que el 4º bit, contando por la derecha, de dicha propiedad, tome el valor **1**.

Para ello utilizaremos el operador **Or** de la siguiente manera:

```
Dispositivo.Config = Dispositivo.Config Or 2^(3)
```

En general para activar el bit número N de un valor M, haremos

```
M = M or 2^(N-1)
```

Operador Xor

El operador **Xor**, realiza una Exclusión lógica entre dos operandos.

Su sintaxis es

resultado **=** *expresión1* **Xor** *expresión2*

Resultado puede ser cualquier variable numérica ó boleana.

Tomará el valor **True**, si una de las dos expresiones es **True** y la otra **False**.

Si cualquiera de ellas es **Null**, dará **Null**.

Si las dos son a la vez **False** o **True** dará **False**.

Cuadro de resultados del operador **Xor**

Expresión 1ª	Expresión 2ª	Resultado
True	True	**False**
True	False	**True**
False	True	**True**
False	False	**False**

En operaciones a nivel de bit, dará 1 si uno de los dos es 0 y el otro 1.

Si los dos bits fueran iguales dará 0.

```
5 Xor 13  →  8
        0101
        1101
        1000
```

El binario **1000** es el número **8**

Supongamos que queremos hacer que el 4º bit, contando por la derecha, de un valor **M** tome el valor **1**.

Esto lo haremos en dos pasos

1. Comprobamos si el valor **M** ya tiene el bit a cero.
2. Si no fuera así lo cambiamos a cero usando **Xor**

```
If M And 8 = 0 then  ' 8 es 2^3 ó lo que es lo mismo 2^(4-1)
  M = M Xor 8
End if
```

Operador Not

El operador **Not**, realiza una Negación lógica sobre una expresión.

Su sintaxis es

resultado **= Not** *expresión*

Resultado puede ser cualquier variable numérica ó boleana.

Tomará el valor **True**, si *expresión* es **False** y a la inversa.

Si *expresión* fuese **Null**, devolverá **Null**.

Cuadro de resultados del operador **Not**

Expresión	Resultado
True	**False**
False	**True**
Null	**Null**

A nivel de bits también se puede utilizar el operador **Not**.

Not 13 → -14

13 0000000000001101

Not 13 **1111111111110010**

El binario **1111111111110010** es el número **-14**

Operador Eqv

El operador **Eqv**, realiza una Equivalencia lógica entre dos expresiones.

Su sintaxis es

resultado **=** *expresión1* **Eqv** *expresión2*

resultado puede ser cualquier variable numérica ó boleana.

Si alguna de las expresiones fuera **Null**, el resultado será **Null**.

Si los dos operandos fuesen **True** ó **False**, daría como resultado **True**.

En caso contrario daría **False**.

Cuadro de resultados del operador **Eqv**

Expresión 1ª	Expresión 2ª	Resultado
True	True	**True**
True	False	**False**
False	True	**False**
False	False	**True**

Podría considerarse como el operador inverso a **Xor**.

En operaciones a nivel de bit, dará 1 si los dos bits fueran iguales.

Si uno de los dos es 0 y el otro 1 dará 0.

```
5 Eqv 13   →   -9
           0000000000000101
           0000000000001101
           1111111111110111
```

El binario **1111111111110111** es el número **-9**

Operador Imp

El operador **Imp**, realiza una Implicación lógica entre dos expresiones.

Su sintaxis es

resultado **=** *expresión1* **Imp** *expresión2*

resultado puede ser cualquier variable numérica ó boleana.

Cuadro de resultados del operador **Imp**

Expresión 1ª	Expresión 2ª	Resultado
True	True	**True**
True	False	**False**
True	Null	**Null**
False	True	**True**
False	False	**True**
False	Null	**True**
Null	True	**True**
Null	False	**Null**
Null	Null	**Null**

De forma equivalente, a nivel de bits los resultados serían

Expresión 1ª	Expresión 2ª	Resultado
0	0	**1**
0	1	**1**
1	0	**0**
1	1	**1**

Prioridad de los operadores

Cuando tenemos una expresión con un operador que une dos operandos, no se plantea problema alguno de interpretación.

Si tenemos

```
intA = 2
intB = 4
intC = INTA + intB
```

La variable **intC** sabemos intuitivamente que tomará el valor **6**.

En expresiones más complejas, puede que no lo tengamos tan claro.

Por ejemplo, si a continuación de ese código hiciéramos:

```
dblD = intA ^ 12 / intC
```

Suponemos que las variables están correctamente declaradas

¿Qué valor tomará **dblD**?

En concreto, **intC** a qué divide, al **12** o al resultado de **intA** elevado a **12**.

La respuesta correcta es la segunda.

Si fuera la primera, `dblD` tomaría el valor `4`.

En este caso toma el valor `682,666666666667`, ya que primero efectúa la potencia, y posteriormente divide el resultado por `12`.

Cada tipo de operador tiene una prioridad en su evaluación.

Si en una misma expresión se incluyen operadores de distintas categorías, primero se evalúan los de más prioridad, y en caso de haber de la misma, se empieza por los que están situados más a la izquierda.

La prioridad en los operadores sigue estos órdenes para cada tipo de operador

Aritméticos	De comparación	Lógicos
Exponenciación (**^**)	Igualdad (**=**)	**Not**
Negación (**–**)	Desigualdad (**<>**)	**And**
Multiplicación y división (*****, **/**)	Menor que (**<**)	**Or**
División de enteros (****)	Mayor que (**>**)	**Xor**
Módulo aritmético (**Mod**)	Menor o igual que (**<=**)	**Eqv**
Adición y substracción (**+**, **–**)	Mayor o igual que (**>=**)	**Imp**
Concatenación de cadenas (**&**)	**Like** **Is**	

Las multiplicaciones y divisiones se evalúan entre sí de izquierda a derecha, siguiendo el orden en el que están escritas.

Esto mismo ocurre con la suma y la resta.

Los operadores aritméticos se evalúan antes que los de comparación.

Todo esto lleva a al conveniencia de la utilización de paréntesis, para tener un perfecto control del orden de interpretación.

En una expresión lo primero que se evalúa son las subexpresiones contenidas dentro de paréntesis, si las hubiera.

En el caso del ejemplo, `dblD = intA ^ 12 / intC` si quisiéramos que primero se efectuara la división de 12 entre `intC` deberíamos haber escrito

```
dblD = intA ^ (12 / intC)
```

En el caso de la expresión:

```
4 Or 19 And 7 Or 13
```

se evalúa en este orden

1. `19 And 7` → `3`
2. `4 Or 3` → `7`
3. `7 Or 13` → `15`

Si quisiéramos que evaluara primero `4 or 19`, `7 or 13` y luego los respectivos resultados mediante And, deberíamos haber escrito

```
(4 Or 19) And (7 Or 13)
```

Que nos dará como resultado 7.

La utilización de paréntesis aclara el código y ayuda a evitar errores en el diseño del mismo, muchas veces con resultados inesperados, y de difícil detección.